
PENNON: Software for linear and nonlinear matrix inequalities

Michal Kočvara¹ and Michael Stingl²

¹ School of Mathematics, University of Birmingham, Birmingham B15 2TT, UK
and Institute of Information Theory and Automation, Academy of Sciences of
the Czech Republic, Pod vodárenskou věží 4, 18208 Praha 8, Czech Republic
`kocvara@maths.bham.ac.uk`

² Institute of Applied Mathematics, University of Erlangen-Nuremberg,
Martensstrasse 3, 91058 Erlangen, Germany `stingl@am.uni-erlangen.de`

1 Introduction

The goal of this paper is to present an overview of the software collection for the solution of linear and nonlinear semidefinite optimization problems PENNON. In the first part we present theoretical and practical details of the underlying algorithm and several implementation issues. In the second part we introduce the particular codes PENSDP, PENBMI and PENNON, focus on some specific features of these codes and show how they can be used for the solution of selected problems.

We use standard notation: \mathbb{S}^m is the space of real symmetric matrices of dimension $m \times m$ and \mathbb{S}_+^m the space of positive semidefinite matrices from \mathbb{S}^m . The inner product on \mathbb{S}^m is defined by $\langle A, B \rangle_{\mathbb{S}^m} := \text{trace}(AB)$. Notation $A \preceq B$ for $A, B \in \mathbb{S}^m$ means that the matrix $B - A$ is positive semidefinite. The norm $\|\cdot\|$ is always the ℓ_2 norm in case of vectors and the spectral norm in case of matrices, unless stated otherwise. Finally, for $\Phi : \mathbb{S}^m \rightarrow \mathbb{S}^m$ and $X, Y \in \mathbb{S}^m$, $D\Phi(X)[Y]$ denotes the directional derivative of Φ with respect to X in direction Y .

2 The main algorithm

2.1 Problem formulation

The nonlinear semidefinite problems can be written in several different ways. In this section, for the sake of simplicity, we will use the following formulation:

$$\begin{aligned}
& \min_{x \in \mathbb{R}^n} f(x) \\
& \text{subject to} \\
& \mathcal{A}(x) \preceq 0.
\end{aligned} \tag{1}$$

Here $f : \mathbb{R}^n \rightarrow \mathbb{R}$ and $\mathcal{A} : \mathbb{R}^n \rightarrow \mathbb{S}^m$ are twice continuously differentiable mappings.

Later in sections on linear SDP, BMI and nonlinear SDP, we will give more specific formulations of the problem. However, the algorithm and theory described in this section applies, with some exceptions discussed later, to all these specific formulations.

2.2 The algorithm

The basic algorithm used in this article is based on the nonlinear rescaling method of R. Polyak [30] and was described in detail in [16] and [31]. Here we briefly recall it and stress points that will be needed in the rest of the paper.

The algorithm is based on the choice of a smooth penalty/barrier function $\Phi_p : \mathbb{S}^m \rightarrow \mathbb{S}^m$ that satisfies a number of assumptions (see [16, 31]) guaranteeing, in particular, that for any $p > 0$

$$\mathcal{A}(x) \preceq 0 \iff \Phi_p(\mathcal{A}(x)) \preceq 0$$

for (at least) all x such that $\mathcal{A}(x) \preceq 0$. Thus for any $p > 0$, problem (1) has the same solution as the following “augmented” problem

$$\begin{aligned}
& \min_{x \in \mathbb{R}^n} f(x) \\
& \text{subject to} \\
& \Phi_p(\mathcal{A}(x)) \preceq 0.
\end{aligned} \tag{2}$$

The Lagrangian of (2) can be viewed as a (generalized) augmented Lagrangian of (1):

$$F(x, U, p) = f(x) + \langle U, \Phi_p(\mathcal{A}(x)) \rangle_{\mathbb{S}^m}; \tag{3}$$

here $U \in \mathbb{S}_+^m$ is a Lagrangian multiplier associated with the inequality constraint.

The algorithm below can be seen as a generalization of the Augmented Lagrangian method.

Algorithm 1. Let x^1 and U^1 be given. Let $p^1 > 0$. For $k = 1, 2, \dots$ repeat until a stopping criterion is reached:

1. $x^{k+1} = \arg \min_{x \in \mathbb{R}^n} F(x, U^k, p^k)$
2. $U^{k+1} = D\Phi_p(\mathcal{A}(x^{k+1}))[U^k]$
3. $p^{k+1} \leq p^k$.

Details of the algorithm, the choice of the penalty function Φ_p , the choice of initial values of x, U and p , the approximate minimization in Step (i) and the update formulas, will be discussed in subsequent sections. The next section concerns the overview of theoretical properties of the algorithm.

2.3 Convergence theory overview

Throughout this section we make the following assumptions on problem (1):

- (A1) $x^* = \arg \min\{f(x)|x \in \Omega\}$ exists, where $\Omega = \{x \in \mathbb{R}^n | \mathcal{A}(x) \preceq 0\}$.
- (A2) The Karush-Kuhn-Tucker necessary optimality conditions hold in x^* , i.e., there exists $U^* \in \mathbb{S}^m$ such that

$$\begin{aligned} f'(x^*) + [\langle U^*, \mathcal{A}_i \rangle]_{i=1}^m &= 0 \\ \langle U^*, \mathcal{A}(x^*) \rangle &= 0 \\ U^* &\succeq 0 \\ \mathcal{A}(x^*) &\preceq 0, \end{aligned} \tag{4}$$

where \mathcal{A}_i denotes the i -th partial derivative of \mathcal{A} at x^* ($i = 1, \dots, n$). Moreover the strict complementary is satisfied.

- (A3) The nondegeneracy condition holds, i.e., if for $1 \leq r < m$ the vectors $s_{m-r+1}, \dots, s_m \in \mathbb{R}^m$ form a basis of the null space of the matrix $\mathcal{A}(x^*)$, then the following set of n -dimensional vectors is linearly independent:

$$v_{i,j} = (s_i^\top \mathcal{A}_1 s_j, \dots, s_i^\top \mathcal{A}_n s_j)^\top, \quad m-r+1 \leq i \leq j \leq m.$$

- (A4) Define $E_0 = (s_{m-r+1}, \dots, s_m)$, where s_{m-r+1}, \dots, s_m are the vectors introduced in assumption (A3). Then the *cone of critical directions* at x^* is defined as

$$\mathcal{C}(x^*) = \left\{ h \in \mathbb{R}^n : \sum_{i=1}^n h_i E_0^\top \mathcal{A}_i E_0 \preceq 0, f'(x^*)^\top h = 0 \right\}.$$

With this the following second order sufficient optimality condition is assumed to hold at (x^*, U^*) : For all $h \in \mathcal{C}(x^*)$ with $h \neq 0$ the inequality

$$h^\top (L''_{xx}(x^*, U^*) + H(x^*, U^*)) h > 0,$$

is satisfied, where L is the classic Lagrangian of (1) defined as

$$L(x, U) = f(x) + \langle U, \mathcal{A}(x) \rangle,$$

$H(x^*, U^*)$ is defined entry-wise by

$$H(x^*, U^*)_{i,j} = -2 \langle U^*, \mathcal{A}_i [\mathcal{A}(x^*)]^\dagger \mathcal{A}_j \rangle \tag{5}$$

(see, for example, [4, p. 490]) and $[\mathcal{A}(x^*)]^\dagger$ is the Moore-Penrose inverse of $\mathcal{A}(x^*)$.

(A5) Let

$$\Omega_p = \{x \in \mathbb{R}^n | \mathcal{A}(x) \preceq pI_m\}.$$

Then the following growth condition holds:

$$\exists \pi > 0 \text{ and } \tau > 0 \text{ such that } \max \{\|\mathcal{A}(x)\| \mid x \in \Omega_\pi\} \leq \tau. \quad (6)$$

Using these assumptions the following local convergence result can be established.

Theorem 1. *Let $\mathcal{A}(x)$ be twice continuously differentiable and assumptions (A1) to (A5) hold for the pair (x^*, U^*) . Then there exists a penalty parameter $p_0 > 0$ large enough and a neighbourhood \mathcal{V} of (x^*, U^*) such that for all $(U, p) \in \mathcal{V}$:*

a) *There exists a vector*

$$\hat{x} = \hat{x}(U, p) = \arg \min \{F(x, U, p) \mid x \in \mathbb{R}^n\}$$

such that $\nabla_x F(\hat{x}, U, p) = 0$.

b) *For the pair \hat{x} and $\hat{U} = \hat{U}(U, p) = D\Phi_p(\mathcal{A}(\hat{x}(U, p)))[U]$ the estimate*

$$\max \left\{ \|\hat{x} - x^*\|, \|\hat{U} - U^*\| \right\} \leq Cp \|U - U^*\| \quad (7)$$

holds, where C is a constant independent of p .

c) *$\hat{x}(U^*, p) = x^*$ and $\hat{U}(U^*, p) = U^*$.*

d) *The function $F(x, U, p)$ is strongly convex with respect to x in a neighborhood of $\hat{x}(U, p)$.*

The proof for Theorem 1 as well as a precise definition of the neighborhood \mathcal{V} is given in [31]. A slightly modified version of Theorem 1 with a particular choice of the penalty function Φ can be found in [21]. An alternative convergence theorem using slightly different assumptions is presented in [29].

An immediate consequence of Theorem 1 is that Algorithm 1 converges with a linear rate of convergence. If $p_k \rightarrow 0$ for $k \rightarrow \infty$ is assumed for the sequence of penalty parameters then the rate of convergence is superlinear.

Remark 1. a) Let x^+ be a local minimum of problem (1) satisfying assumptions (A2) to (A5) and denote by U^+ the corresponding (unique) optimal multiplier. Assume further that there exists a neighborhood S_ν of x^+ such that there is no further first order critical point $\tilde{x} \neq x^+$ in $S_\nu(x^+)$. Then all statements of Theorem 1 remain valid, if we replace (x^*, U^*) by (x^+, U^+) and the function $\hat{x}(U, p)$ by

$$\hat{x}_{\text{loc}}(U, p) = \arg \min \{F(x, U, p) \mid x \in \mathbb{R}^n, x \in S_\nu\}. \quad (8)$$

Moreover Theorem 1 d) guarantees that $F(x, U, p)$ is strongly convex in a neighborhood of x^+ for all $(U, p) \in \mathcal{V}$. Consequently any local descent method applied to the problem

$$(i') \quad \text{Find } x^{k+1} \text{ such that } \|\nabla_x F(x, U^k, p^k)\| = 0 \quad (9)$$

will automatically find a solution, which satisfies the additional constraint $x^{k+1} \in S_\nu$ provided it is started with x^k close enough to x^+ . Moreover, Algorithm 1 will converge to the local optimum x^+ (see [31] for more details).

b) A global convergence result can be found in [31].

2.4 Choice of Φ_p

The penalty function Φ_p of our choice is defined as follows:

$$\Phi_p(\mathcal{A}(x)) = -p^2(\mathcal{A}(x) - pI)^{-1} - pI. \quad (10)$$

The advantage of this choice is that it gives closed formulas for the first and second derivatives of Φ_p . Defining

$$\mathcal{Z}(x) = -(\mathcal{A}(x) - pI)^{-1} \quad (11)$$

we have (see [16]):

$$\frac{\partial}{\partial x_i} \Phi_p(\mathcal{A}(x)) = p^2 \mathcal{Z}(x) \frac{\partial \mathcal{A}(x)}{\partial x_i} \mathcal{Z}(x) \quad (12)$$

$$\begin{aligned} \frac{\partial^2}{\partial x_i \partial x_j} \Phi_p(\mathcal{A}(x)) = p^2 \mathcal{Z}(x) & \left(\frac{\partial \mathcal{A}(x)}{\partial x_i} \mathcal{Z}(x) \frac{\partial \mathcal{A}(x)}{\partial x_j} + \frac{\partial^2 \mathcal{A}(x)}{\partial x_i \partial x_j} \right. \\ & \left. + \frac{\partial \mathcal{A}(x)}{\partial x_j} \mathcal{Z}(x) \frac{\partial \mathcal{A}(x)}{\partial x_i} \right) \mathcal{Z}(x). \end{aligned} \quad (13)$$

2.5 The modified Newton method

To solve the (possibly nonconvex) unconstrained minimization problem in Step 1, we use the following modification of the Newton method with line-search:

Algorithm 2. Given an initial iterate x_0 , repeat for all $k = 0, 1, 2, \dots$ until a stopping criterion is reached

1. Compute the gradient g_k and Hessian H_k of F at x_k .
2. Try to factorize H_k by Cholesky decomposition. If H_k is factorizable, set $\hat{H} = H_k$ and go to Step 4.
3. Compute $\beta \in [-\lambda_{\min}, -2\lambda_{\min}]$, where λ_{\min} is the minimal eigenvalue of H_k and set $\hat{H} = H_k + \beta I$.
4. Compute the search direction $d_k = -\hat{H}^{-1}g_k$.
5. Perform line-search in direction d_k . Denote the step-length by s_k .
6. Set $x_{k+1} = x_k + s_k d_k$.

The step-length s in direction d is calculated by a gradient free line-search algorithm that tries to satisfy the Armijo condition. Obviously, for a convex F , Algorithm 2 is just the damped Newton method, which is known to converge under standard assumptions.

If, in the non-convex case, the Cholesky factorization in Step 2 fails, we calculate the value of β in Step 3 in the following way:

Algorithm 3. For a given $\beta_0 > 0$

1. Set $\beta = \beta_0$.
2. Try to factorize $H + \beta I$ by the Cholesky method.
3. If the factorization fails due to a negative pivot element, go to step 4, otherwise go to step 5.
4. If $\beta \geq \beta_0$, set $\beta = 2\beta$ and continue with 2. Otherwise go to step 6.
5. If $\beta \leq \beta_0$, set $\beta = \frac{\beta}{2}$ and continue with step 2. Otherwise STOP.
6. Set $\beta = 2\beta$ and STOP.

Obviously, when Algorithm 3 terminates we have $\beta \in [-\lambda_{\min}, -2\lambda_{\min}]$. It is well known from the nonlinear programming literature that under quite mild assumptions any cluster point of the sequence generated by Algorithm 2 is a first order critical point of problem in Step 1 of Algorithm 1.

Remark 2. There is one exception, when we use a different strategy for the calculation of β . The exception is motivated by the observation that the quality of the search direction gets poor, if we choose β too close to $-\lambda_{\min}$. Therefore, if we encounter bad quality of the search direction, we use a bisection technique to calculate an approximation of λ_{\min} , denoted by λ_{\min}^a , and replace β by $-1.5\lambda_{\min}^a$.

Remark 3. Whenever we will speak about the Newton method or Newton system, later in the paper, we will always have in mind the modified method described above.

2.6 How to solve the linear systems?

In both algorithms proposed in the preceding sections one has to solve repeatedly linear systems of the form

$$(H + D)d = -g, \quad (14)$$

where D is a diagonal matrix chosen such that the matrix $H + D$ is positive definite. There are two categories of methods, which can be used to solve problems of type (14): direct and iterative methods. Let us first concentrate on the direct methods.

Cholesky method

Since the system matrix in (14) is always positive definite, our method of choice is the Cholesky method. Depending on the sparsity structure of H , we use two different realizations:

- If the fill-in of the Hessian is below 20%, we use a sparse Cholesky solver which is based on ideas of Ng and Peyton [27]. The solver makes use of the fact that the sparsity structure is the same in each Newton step in all iterations. Hence the sparsity pattern of H , reordering of rows and columns to reduce the fill-in in the Cholesky factor, and symbolic factorization of H are all performed just once at the beginning of Algorithm 1. Then, each time the system (14) has to be solved, the numeric factorization is calculated based on the precalculated symbolic factorization. Note that we added stabilization techniques described in [34] to make the solver more robust for almost singular system matrices.
- Otherwise, if the Hessian is dense, we use the ATLAS implementation of the LAPACK Cholesky solver DPOTRF.

Iterative methods

We solve the system $\hat{H}d = -g$ with a symmetric positive definite and, possibly, ill-conditioned matrix $\hat{H} = H + D$. We use the very standard preconditioned conjugate gradient method. The algorithm is well known and we will not repeat it here. The algorithm is stopped when the normalized residuum is sufficiently small:

$$\|\hat{H}d_k + g\|/\|g\| \leq \epsilon.$$

In our tests, the choice $\epsilon = 5 \cdot 10^{-2}$ was sufficient.

Preconditioners

We are looking for a preconditioner—a matrix $M \in \mathbb{S}_+^n$ —such that the system $M^{-1}\hat{H}d = -M^{-1}g$ can be solved more efficiently than the original system $\hat{H}d = -g$. Apart from standard requirements that the preconditioner should be efficient and inexpensive, we also require that it should only use Hessian-vector products. This is particularly important in the case when we want to use the Hessian-free version of the algorithm.

Diagonal preconditioner

This is a simple and often-used preconditioner with

$$M = \text{diag}(\hat{H}).$$

On the other hand, being simple and general, it is not considered to be very efficient. Furthermore, we need to know the diagonal elements of the Hessian.

It is certainly possible to compute these elements by Hessian-vector products. For that, however, we would need n gradient evaluations and the approach would become too costly.

L-BFGS preconditioner

Introduced by Morales-Nocedal [25], this preconditioner is intended for application within the Newton method. (In a slightly different context, the (L-)BFGS preconditioner was also proposed in [10].) The algorithm is based on the limited-memory BFGS formula ([28]) applied to successive CG (instead of Newton) iterations. The preconditioner, as used in PENNON is described in detail in [18]. Here we only point out some important features.

As recommended in the standard L-BFGS method, we used 16–32 correction pairs, if they were available. Often the CG method finished in less iterations and in that case we could only use the available iterations for the correction pairs. If the number of CG iterations is higher than the required number of correction pairs μ , we may ask how to select these pairs. We have two options: Either we take the last μ pairs or an “equidistant” distribution over all CG iterations. The second option is slightly more complicated but we may expect it to deliver better results.

The L-BFGS preconditioner has the big advantage that it only needs Hessian-vector products and can thus be used in the Hessian-free approaches. On the other hand, it is more complex than the above preconditioners; also our results are not conclusive concerning the efficiency of this approach. For many problems it worked satisfactorily, for some, on the other hand, it even lead to higher number of CG steps than without preconditioner.

2.7 Multiplier and penalty update

For the penalty function Φ_p from (10), the formula for update of the matrix multiplier U in Step (ii) of Algorithm 1 reduces to

$$U^{k+1} = (p^k)^2 \mathcal{Z}(x^{k+1}) U^k \mathcal{Z}(x^{k+1}) \quad (15)$$

with \mathcal{Z} defined as in (11). Note that when U^k is positive definite, so is U^{k+1} . We set U^1 equal to a positive multiple of the identity.

Numerical tests indicate that big changes in the multipliers should be avoided for the following reasons. Big change of U means big change of the augmented Lagrangian that may lead to a large number of Newton steps in the subsequent iteration. It may also happen that already after few initial steps the multipliers become ill-conditioned and the algorithm suffers from numerical difficulties. To overcome these, we do the following:

1. Calculate U^{k+1} using (15).
2. Choose a positive $\mu_A \leq 1$, typically 0.5.
3. Compute $\lambda_A = \min \left(\mu_A, \mu_A \frac{\|U^k\|_F}{\|U^{k+1} - U^k\|_F} \right)$.

4. Update the current multiplier by

$$U^{new} = U^k + \lambda_A(U^{k+1} - U^k).$$

Given an initial iterate x^1 , the initial penalty parameter p^1 is chosen large enough to satisfy the inequality

$$p^1 I - \mathcal{A}(x^1) \succ 0.$$

Let $\lambda_{\max}(\mathcal{A}(x^{k+1})) \in (-\infty, p^k)$ denote the maximal eigenvalue of $\mathcal{A}(x^{k+1})$, $\pi < 1$ be a constant factor, depending on the initial penalty parameter p^1 (typically chosen between 0.3 and 0.6) and x_{feas} be a feasible point. Let l be set to 0 at the beginning of Algorithm 1. Using these quantities, our strategy for the penalty parameter update can be described as follows:

1. If $p^k < p_{\text{eps}}$, set $\gamma = 1$ and go to 6.
2. Calculate $\lambda_{\max}(\mathcal{A}(x^{k+1}))$.
3. If $\pi p^k > \lambda_{\max}(\mathcal{A}(x^{k+1}))$, set $\gamma = \pi$, $l = 0$ and go to 6.
4. If $l < 3$, set $\gamma = (\lambda_{\max}(\mathcal{A}(x^{k+1})) + p^k) / (2p^k)$, set $l := l + 1$ and go to 6.
5. Let $\gamma = \pi$, find $\lambda \in (0, 1)$ such, that

$$\lambda_{\max}(\mathcal{A}(\lambda x^{k+1} + (1 - \lambda)x_{\text{feas}})) < \pi p^k,$$

set $x^{k+1} = \lambda x^{k+1} + (1 - \lambda)x_{\text{feas}}$ and $l := 0$.

6. Update current penalty parameter by $p^{k+1} = \gamma p^k$.

The reasoning behind steps 3 to 5 is as follows: As long as the inequality

$$\lambda_{\max}(\mathcal{A}(x^{k+1})) < \pi p^k \tag{16}$$

holds, the values of the augmented Lagrangian in the next iteration remain finite and we can reduce the penalty parameter by the predefined factor π . As soon as inequality (16) is violated, an update using π would result in an infinite value of the augmented Lagrangian in the next iteration. Therefore the new penalty parameter should be chosen from the interval $(\lambda_{\max}(\mathcal{A}(x^{k+1})), p^k)$. Because a choice close to the left boundary of the interval leads to large values of the augmented Lagrangian, while a choice close to the right boundary slows down the algorithm, we choose γ such that

$$p^{k+1} = \frac{\lambda_{\max}(\mathcal{A}(x^{k+1})) + p^k}{2}.$$

In order to avoid stagnation of the penalty parameter update process due to repeated evaluations of step 4, we redefine x^{k+1} using the feasible point x_{feas} whenever step 4 is executed in three successive iterations; this is controlled by the parameter l . If no feasible point is yet available, Algorithm 1 is stopped and restarted from the scratch with a different choice of initial multipliers. The parameter p_{eps} is typically chosen as 10^{-6} . In case we detect problems with convergence of Algorithm 1, p_{eps} is decreased and the penalty parameter is updated again, until the new lower bound is reached.

2.8 Initialization and stopping criteria

Initialization

Algorithm 1 can start with an arbitrary primal variable $x \in \mathbb{R}^n$. Therefore we simply choose $x^1 = 0$. For the description of the multiplier initialization strategy we rewrite problem (SDP) in the following form:

$$\begin{aligned} & \min_{x \in \mathbb{R}^n} f(x) \\ & \text{subject to} \\ & \mathcal{A}_i(x) \preceq 0, \quad i = 1, \dots, \sigma. \end{aligned} \quad (17)$$

Here $\mathcal{A}_i(x) \in \mathbb{S}^{m_j}$ are diagonal blocks of the original constrained matrix $\mathcal{A}(x)$ and we have $\sigma = 1$ if $\mathcal{A}(x)$ consists of only one block. Now the initial values of the multipliers are set to

$$U_j^1 = \mu_j I_{m_j}, \quad j = 1, \dots, \sigma,$$

where I_{m_j} are identity matrices of order m_j and

$$\mu_j = m_j \max_{1 \leq \ell \leq n} \frac{1 + \left| \frac{\partial f(x)}{\partial x_\ell} \right|}{1 + \left\| \frac{\partial \mathcal{A}(x)}{\partial x_\ell} \right\|}. \quad (18)$$

Given the initial iterate x^1 , the initial penalty parameter p^1 is chosen large enough to satisfy the inequality

$$p^1 I - \mathcal{A}(x^1) \succ 0.$$

Stopping criterion in the sub-problem

In the first iterations of Algorithm 1, the approximate minimization of F is stopped when $\left\| \frac{\partial}{\partial x} F(x, U, p) \right\| \leq \alpha$, where $\alpha = 0.01$ is a good choice in most cases. In the remaining iterations, after a certain precision is reached, α is reduced in each outer iteration by a constant factor, until a certain $\underline{\alpha}$ (typically 10^{-7}) is reached.

Stopping criterion for the main algorithm

We have implemented two different stopping criteria for the main algorithm.

- *First alternative:* The main algorithm is stopped if both of the following inequalities hold:

$$\frac{|f(x^k) - F(x^k, U^k, p)|}{1 + |f(x^k)|} < \varepsilon_1, \quad \frac{|f(x^k) - f(x^{k-1})|}{1 + |f(x^k)|} < \varepsilon_1,$$

where ε_1 is typically 10^{-7} .

- *Second alternative:* The second stopping criterion is based on the KKT-conditions. Here the algorithm is stopped, if

$$\min \{ \lambda_{\max}(\mathcal{A}(x)), |\langle \mathcal{A}(x), U \rangle|, \|\nabla_x F(x, U, p)\| \} \leq \varepsilon_2.$$

2.9 Complexity

The computational complexity of Algorithm 1 is clearly dominated by Step 1. In each step of the Newton method, there are two critical issues: assembling of the Hessian of the augmented Lagrangian and solution of the linear system of equations (the Newton system).

Hessian assembling

Full matrices

Assume first that all the data matrices are full. The assembling of the Hessian (13) can be divided into the following steps:

- Calculation of $\mathcal{Z}(x) \rightarrow O(m^3 + m^2n)$.
- Calculation of $\mathcal{Z}(x)U\mathcal{Z}(x) \rightarrow O(m^3)$.
- Calculation of $\mathcal{Z}(x)U\mathcal{Z}(x)\mathcal{A}'_i(x)\mathcal{Z}(x)$ for all $i \rightarrow O(m^3n)$.
- Assembling the rest $\rightarrow O(m^2n^2)$.

Now it is straightforward to see that an estimate of the complexity of assembling of (13) is given by $O(m^3n + m^2n^2)$.

Many optimization problems, however, have very sparse data structure and therefore have to be treated by sparse linear algebra routines. We distinguish three basic types of sparsity.

The block diagonal case

The first case under consideration is the block diagonal case. In particular, we want to describe the case, where

- the matrix $\mathcal{A}(x)$ consists of many (small) blocks.

In this situation the original SDP problem (1) can be written in the form (17). If we define $\bar{m} = \max\{m_i \mid i = 1, \dots, d\}$ we can estimate the computational complexity of the Hessian assembling by $O(d\bar{m}^3n + d\bar{m}^2n^2)$. An interesting subcase of problem (17) is when

- each of the matrix constraints $\mathcal{A}_i(x)$ involves just a few components of x .

If we denote the maximal number of components of x on which each of the blocks $\mathcal{A}_i(x), i = 1, 2, \dots, d$ depends by \bar{n} , our complexity formula becomes $O(d\bar{m}^3\bar{n} + d\bar{m}^2\bar{n}^2)$. If we further assume that the numbers \bar{n} and \bar{m} are of order $O(1)$, then the complexity estimate can be further simplified to $O(d)$. A typical example for this sparsity class are the ‘mater’ problems discussed in section 3.

The case when $\mathcal{A}(x)$ is dense and $\mathcal{A}'_i(x)$ are sparse

Let us first mention that for any index pair $(i, j) \in \{1, \dots, n\} \times \{1, \dots, n\}$ the non-zero structure of the matrix $\mathcal{A}''_{i,j}(x)$ is given by (a subset of the) intersection of the non-zero index sets of the matrices $\mathcal{A}'_i(x)$ and $\mathcal{A}'_j(x)$. We assume that

- there are at most $O(1)$ non-zero entries in $\mathcal{A}'_i(x)$ for all $i = 1, \dots, n$.

Then the calculation of the term

$$[\langle \mathcal{Z}(x)U\mathcal{Z}(x), \mathcal{A}''_{i,j}(x) \rangle]_{i,j=1}^n$$

can be performed in $O(n^2)$ time. In the paper by Fujisawa, Kojima and Nakata on exploiting sparsity in semidefinite programming [8] several ways are presented how to calculate a matrix of the form

$$D_1 S_1 D_2 S_2 \tag{19}$$

efficiently, if D_1 and D_2 are dense and S_1 and S_2 are sparse matrices. If our assumption above holds, the calculation of the matrix

$$[\langle \mathcal{Z}(x)U\mathcal{Z}(x)\mathcal{A}'_j(x)\mathcal{Z}(x), \mathcal{A}'_i(x) \rangle]_{i,j=1}^n$$

can be performed in $O(n^2)$ time. Thus, recalling that for the calculation of $\mathcal{Z}(x)$ we have to compute the inverse of an $(m \times m)$ -matrix, we get the following complexity estimate for the Hessian assembling: $O(m^3 + n^2)$. Note that in our implementation we follow the ideas presented in [8]. Many linear SDP problems coming from real world applications have exactly the sparsity structure discussed in this paragraph.

The case when $\mathcal{A}(x)$ and the Cholesky factor of $\mathcal{A}(x)$ is sparse

Also in this case we can conclude that all partial derivatives of $\mathcal{A}(x)$ of first and second order are sparse matrices. Therefore it suffices to assume that

- the matrix $\mathcal{A}(x)$ has at most $O(1)$ non-zero entries.

We have to compute expressions of type

$$(\mathcal{A}(x) - pI)^{-1}U(\mathcal{A}(x) - pI)^{-1} \quad \text{and} \quad (\mathcal{A}(x) - pI)^{-1}.$$

Note that each of the matrices above can be calculated by maximally two operations of the type $(A - I)^{-1}M$, where M is a symmetric matrix. Now assume that not only $\mathcal{A}(x)$ but also its Cholesky factor is sparse. Then, obviously, the Cholesky factor of $(\mathcal{A}(x) - pI)$, denoted by L , will also be sparse. This leads to the following assumption:

- Each column of L has at most $O(1)$ non-zero entries.

Now the i -th column of $C := (\mathcal{A}(x) - pI)^{-1}M$ can then be computed as

$$C^i = (L^{-1})^T L^{-1} M^i, \quad i = 1, \dots, n,$$

and the complexity of computing C by Cholesky factorization is $O(m^2)$, compared to $O(m^3)$ when computing the inverse of $(\mathcal{A}(x) - pI)$ and its multiplication by U . So the overall complexity of Hessian assembling is of order $O(m^2 + n^2)$.

Remark 4. Recall that in certain cases, we do not need to assemble the Hessian matrix. In this case the complexity estimates can be improved significantly; see the next section.

Solution of the Newton system

As mentioned above, the Newton system

$$Hd = -g \tag{20}$$

can either be solved by a direct (Cholesky) solver or by an iterative method.

Cholesky method

The complexity of Cholesky algorithm is $O(n^3)$ for dense matrices and $O(n^\kappa)$, $1 \leq \kappa \leq 3$ for sparse matrices, where κ depends on the sparsity structure of the matrix, going from a diagonal to a full matrix.

Iterative algorithms

From the complexity viewpoint, the only demanding step in the CG method is a matrix-vector product with a matrix of dimension n . For a dense matrix and vector, it needs $O(n^2)$ operations. Theoretically, in exact arithmetics, the CG method needs n iterations to find an exact solution of the system, hence it is equally expensive as the Cholesky algorithm. There are, however, two points that may favor the CG method.

First, it is well known that the convergence behavior of the CG method can be significantly improved by preconditioning. The choice of the preconditioner M will be the subject of the next section.

The second—and very important—point is that we actually do not need an exact solution of the Newton system. On the contrary, a rough approximation of it will do (see [11, Thm. 10.2]). Hence, in practice, we may need just a few CG iterations to reach the required accuracy. This is in contrast with the Cholesky method where we cannot control the accuracy of the solution and always have to compute the exact one (within the machine precision). Note that we always start the CG method with initial approximation $d_0 = 0$; thus, performing just one CG step, we would obtain the steepest descend

method. Doing more steps, we improve the search direction toward the Newton direction; note the similarity to the Toint-Steihaug method [28].

Summarizing these two points: when using the CG algorithm, we may expect to need just $O(n^2)$ operations, at least for well-conditioned (or well-preconditioned) systems.

Note that we are still talking about dense problems. The use of the CG method is a bit nonstandard in this context—usually it is preferable for large sparse problems. However, due to the fact that we just need a very rough approximation of the solution, we may favor it to the Cholesky method also for medium-sized dense problems.

Approximate Hessian formula

When solving the Newton system by the CG method, the Hessian is only needed in a matrix-vector product of the type $Hv := \nabla^2 F(x^k)v$. Because we only need to compute the products, we may use a finite difference formula for the approximation of this product

$$\nabla^2 F(x^k)v \approx \frac{\nabla F(x^k + hv) - \nabla F(x^k)}{h} \quad (21)$$

with $h = (1 + \|x^k\|_2\sqrt{\varepsilon})$; see [28]. In general, ε is chosen so that the formula is as accurate as possible and still not influenced by round-off errors. The “best” choice is obviously case dependent; in our implementation, we use $\varepsilon = 10^{-6}$. Hence the complexity of the CG method amounts to the number of CG iterations times the complexity of gradient evaluation, which is of order $O(m^3 + Kn)$, where K denotes the maximal number of nonzero entries in $\mathcal{A}'_i(x)$, $i = 1, 2, \dots, n$. This may be in sharp contrast with the Cholesky method approach when we have to compute the full Hessian *and* solve the system by Cholesky method. Again, we have the advantage that we do not have to store the Hessian in the memory.

This approach is clearly not always applicable. With certain SDP problems it may happen that the Hessian computation is not much more expensive than the gradient evaluation. In this case the Hessian-free approach may be rather time-consuming. Indeed, when the problem is ill-conditioned and we need many CG iterations, we have to evaluate the gradient many (thousand) times. On the other hand, when using Cholesky method, we compute the Hessian just once.

3 PENSDP

When both functions in (1) are linear, the problem simplifies to a standard (primal or dual, as you like) linear semidefinite programming problem (LSDP)

$$\begin{aligned}
 & \min_{x \in \mathbb{R}^n} f^T x \\
 & \text{subject to} \\
 & \sum_{k=1}^n x_k A_k - A_0 \preceq 0.
 \end{aligned} \tag{22}$$

Here we write explicitly all matrix inequality constraints, as well as linear constraints, in order to introduce necessary notation. In the next sections, we will present some special features of the code PENSDP designed to solve (22), as well as selected numerical examples demonstrating its capabilities.

3.1 The code PENSDP

Special features

Stopping criteria

In the case of linear semidefinite programs, we have additionally adopted the DIMACS criteria [24]. To define these criteria, we denote $\tilde{\mathcal{A}}(x) = \sum_{k=1}^n x_k A_k$. Recall that U is the corresponding Lagrangian multiplier and let $\tilde{\mathcal{A}}^*(\cdot)$ denote the adjoint operator to $\tilde{\mathcal{A}}(\cdot)$. The DIMACS error measures are defined as

$$\begin{aligned}
 \text{err}_1 &= \frac{\|\tilde{\mathcal{A}}^*(U) - f\|}{1 + \|f\|} \\
 \text{err}_2 &= \max \left\{ 0, \frac{-\lambda_{\min}(U)}{1 + \|f\|} \right\} & \text{err}_4 &= \max \left\{ 0, \frac{-\lambda_{\min}(\tilde{\mathcal{A}}(x) - A_0)}{1 + \|A_0\|} \right\} \\
 \text{err}_5 &= \frac{\langle A_0, U \rangle - f^T x}{1 + |\langle A_0, U \rangle| + |f^T x|} & \text{err}_6 &= \frac{\langle \tilde{\mathcal{A}}(x) - A_0, U \rangle}{1 + |\langle A_0, U \rangle| + |f^T x|}.
 \end{aligned}$$

Here, err_1 represents the (scaled) norm of the gradient of the Lagrangian, err_2 and err_4 is the dual and primal infeasibility, respectively, and err_5 and err_6 measure the duality gap and the complementarity slackness. Note that, in our code, $\text{err}_2 = 0$ by definition; also err_3 that involves the slack variable (not used in our problem formulation) is automatically zero. If the “DIMACS stopping criterion” is activated we require that

$$\text{err}_k \leq \delta_{\text{DIMACS}}, \quad k \in \{1, 4, 5, 6\}.$$

Implicit Hessian formula

As mentioned before, when solving the Newton system by the CG method, the Hessian is only needed in a matrix-vector product of the type $Hv := \nabla^2 F(x^k)v$. Instead of computing the Hessian matrix explicitly and then multiplying it by a vector v , we can use the following formula for the Hessian-vector multiplication

$$\nabla^2 F(x^k)v = 2\mathcal{A}^* \left((p^k)^2 \mathcal{Z}(x^k) U^k \mathcal{Z}(x^k) \mathcal{A}(v) \mathcal{Z}(x^k) \right), \quad (23)$$

where we assume that \mathcal{A} is linear of the form $A(x) = \sum_{i=1}^n x_i A_i$ and \mathcal{A}^* denotes its adjoint. Hence, in each CG step, we only have to evaluate matrices $\mathcal{A}(v)$ (which is simple), $\mathcal{Z}(x^k)$ and $\mathcal{Z}(x^k) U^k \mathcal{Z}(x^k)$ (which are needed in the gradient computation, anyway), and perform two additional matrix-matrix products. The resulting complexity formula for one Hessian-vector product is thus $O(m^3 + Kn)$, where again K denotes the maximal number of nonzero entries in $\mathcal{A}'_i(x)$, $i = 1, 2, \dots, n$.

The additional (perhaps the main) advantage of this approach is the fact that we do not have to store the Hessian in the memory, thus the memory requirements (often the real bottleneck of SDP codes) are drastically reduced.

Dense versus sparse

For the efficiency of PENSDP, it is important to know if the problem has a sparse or dense Hessian. The program can check this automatically. The check, however, may take some time and memory, so if the user knows that the Hessian is dense (and this is the case of most problems), this check can be avoided. This, for certain problems, can lead to substantial savings not only in CPU time but also in memory requirements.

Hybrid mode

For linear semidefinite programming problems, we use the following hybrid approach, whenever the number of variables n is large compared to the size of the matrix constraint m : We try to solve the linear systems using the iterative approach as long as the iterative solver needs a moderate number of iterations. In our current implementation the maximal number of CG iterations allowed is 100. Each time the maximal number of steps is reached, we solve the system again by the Cholesky method. Once the system is solved by the Cholesky method, we use the Cholesky factor as a preconditioner for the iterative solver in the next system. As soon as the iterative solver fails three times in sequel, we completely switch to the Cholesky method.

The hybrid mode allows us to reach a high precision solution while keeping the solution time low. The main reason is that, when using the iterative approach, the Hessian of the Augmented Lagrangian has not to be calculated explicitly.

User interfaces

The user has a choice of several interfaces to PENSDP.

SDPA interface

The problem data are written in an ASCII input file in a SDPA sparse format, as introduced in [9]. The code needs an additional ASCII input file with parameter values.

C/C++/FORTRAN interface

PENSDP can also be called as a function (or subroutine) from a C, C++ or FORTRAN program. In this case, the user should link the PENSDP library to his/her program. In the program the user then has to specify problem dimensions, code parameters and the problem data (vectors and matrices) in a sparse format.

MATLAB interface

In MATLAB, PENSDP is called with the following arguments:

```
[f,x,u,iflag,niter,feas] = pensdpm(pen);
```

where `pen` a MATLAB structure array with fields describing the problem dimensions and problem data, again in a sparse format.

YALMIP interface

The most comfortable way of preparing the data and calling PENSDP is via YALMIP [22]. YALMIP is a modelling language for advanced modeling and solution of convex and nonconvex optimization problems. It is implemented as a free toolbox for MATLAB. When calling PENSDP from YALMIP, the user does not have to bother with the sparsity pattern of the problem—any linear optimization problem with vector or matrix variables will be translated by YALMIP into PENSDP data structure.

3.2 Numerical experiments

It is not our goal to compare PENSDP with other linear SDP solvers. This is done elsewhere in this book and the reader can also consult the benchmark page of Hans Mittelmann³, containing contemporary results. We will thus present only results for selected problems and will concentrate on the effect of special features available in PENSDP. The results for the ‘mater’ and ‘rose13’ problems were obtained on an Intel Core i7 processor 2.67GHz with 4GB memory.

Sparsity: ‘mater’ problems

Let us consider the ‘mater*’ problems from Mittelmann’s collection⁴. These problems are significant by several different sparsity patterns of the problem data. The problem has many small matrix constraints, the data matrices are sparse, only very few variables are involved in each constraint and the resulting Hessian matrix is sparse. We cannot switch off sparsity handling in

³ plato.la.asu.edu/bench.html

⁴ plato.asu.edu/ftp/sparse_sdp.html

routines for Hessian assembling but we can run the code with (forced use of) dense Cholesky factorization and with sparse Cholesky routine. For instance, problem ‘mater3’ with 1439 variables and 328 matrix constraints of size 11 was solved in 32 seconds using the dense Cholesky and only 4 seconds using the sparse Cholesky routine. The difference is, of course, more dramatic for larger problems. The next problem ‘mater4’ has 4807 variables and 1138 matrix constraints of size 11. While the sparse version of PENS DP only needed 20 seconds to solve it, the dense version needed 1149 second. And while the largest problem ‘mater6’ (20463 variables and 4968 matrix constraints) does not even fit in the 4GB memory for the dense version, the sparse code needs only 100MB and solves the problem in 134 seconds.

Iterative solver: ‘TOH’ collection

The effect of the use of preconditioned conjugate gradient method for the solution of the Newton system was described in detail in [18, 19]. Recall that iterative solvers are suitable for problems with a large number of variable and relatively small constraint matrices. We select from [18, 19] two examples arising from maximum clique problems on randomly generated graphs (the ‘TOH’ collection in [18]). The first example is ‘theta62’ with 13390 variables and matrix size 300. This problem could still be solved using the direct (dense Cholesky) solver and the code needed 13714 seconds to solve it. Compared to that, the iterative version of the code only needed 40 seconds to obtain the solution with the same precision. The average number of CG steps in each Newton system was only 10. The largest problem solved in the paper was ‘theta162’ with 127600 variables and a matrix constraint of size 800. Note that the Hessians of this example is *dense*, so to solve the problem by the direct version of PENS DP (or by any other interior-point algorithm) would mean to store and factorize a full matrix of dimension 127600 by 127600. On the other hand, the iterative version of PENS DP, being effectively a first-order code, has only modest memory requirements and allowed us to solve this problem in only 672 seconds.

Hybrid mode: ‘rose13’

To illustrate the advantages of the hybrid mode, we consider the problem ‘rose13’ from Mittelmann’s collection⁵. The problem has 2379 variables and one matrix constraint of size 105. When we solve the problem by PENS DP with a direct solver of the Newton system, the code needs 17 global iterations, 112 Newton steps and the solution is obtained in 188 seconds CPU time, 152 seconds of which is spent in the Cholesky factorization routine.

Let us now solve the problem using the iterative solver for the Newton systems. Below we see the first and the last iterations of PENS DP. The required precision of DIMACS criteria is $\delta_{\text{DIMACS}} = 10^{-3}$.

⁵ plato.asu.edu/ftp/sparse_sdp.html

```

*****
* it |      obj      |      opt      |      Nwt |      CG      *
*****
|  0|  0.0000e+000 |  0.0000e+000 |      0 |      0 |
|  1|  1.8893e+003 |  8.3896e+000 |     10 |     321 |
|  2|  2.2529e+002 |  8.2785e+000 |     17 |    1244 |
...
|  9| -1.1941e+001 |  2.2966e+000 |     36 |    9712 |
| 10| -1.1952e+001 |  4.9578e+000 |     46 |   10209 |
...
| 15| -1.1999e+001 |  5.0429e-002 |    119 |   103905 |
| 16| -1.1999e+001 |  4.4050e-003 |    134 |   167186 |
*****
    
```

The table shows the global iterations of Algorithm 1, the value of the objective function and the gradient of the augmented Lagrangian and, in the last two columns, the cumulative number of Newton and CG steps. The code needed a large number of CG steps that was growing with increasing conditioning of the Newton system. The problem was solved in 732 seconds of CPU time. When we try to solve the same problem with a higher precision ($\delta_{\text{DIMACS}} = 10^{-7}$), the iterative method, and consequently the whole algorithm, will get into increasing difficulties. Below we see the last two iterations of PENSDP before it was stopped due to one-hour time limit.

```

...
| 28| -1.2000e+001 |  5.2644e-002 |    373 |   700549 |
| 29| -1.2000e+001 |  3.0833e-003 |    398 |   811921 |
*****
    
```

We can see that the optimality criterium is actually oscillating around 10^{-3} .

We now switch the hybrid mode on. The difference will be seen already in the early iterations of PENSDP. Running the problem with $\delta_{\text{DIMACS}} = 10^{-3}$, we get the following output

```

*****
* it |      obj      |      opt      |      Nwt |      CG      *
*****
|  0|  0.0000e+000 |  0.0000e+000 |      0 |      0 |
|  1|  1.8893e+003 |  8.3896e+000 |     10 |     321 |
|  2|  2.3285e+002 |  1.9814e+000 |     18 |     848 |
...
|  9| -1.1971e+001 |  4.5469e-001 |     36 |    1660 |
| 10| -1.1931e+001 |  7.4920e-002 |     63 |    2940 |
...
| 13| -1.1998e+001 |  4.1400e-005 |    104 |    5073 |
| 14| -1.1999e+001 |  5.9165e-004 |    115 |    5518 |
*****
    
```

The CPU time needed was only 157 seconds, 130 of which were spent in the Cholesky factorization routine. When we now increase the precision to $\delta_{\text{DIMACS}} = 10^{-7}$, the PENSDP with the hybrid mode will only need a few more iterations to reach it:

```

...
| 16| -1.2000e+001 | 9.8736e-009 | 142 | 6623 |
| 17| -1.2000e+001 | 5.9130e-007 | 156 | 7294 |
*****

```

The total CPU time increased to 201 seconds, 176 of which were spent in Cholesky factorization.

Notice that the difference between the direct solver and the hybrid method would be even more significant for larger problems, such as ‘rose15’.

4 PENBMI

We solve the SDP problem with quadratic objective function and linear and bilinear matrix inequality constraints:

$$\min_{x \in \mathbb{R}^n} \frac{1}{2} x^T Q x + f^T x \quad (24)$$

subject to

$$\sum_{k=1}^n b_k^i x_k \leq c^i, \quad i = 1, \dots, N_\ell$$

$$A_0^i + \sum_{k=1}^n x_k A_k^i + \sum_{k=1}^n \sum_{\ell=1}^n x_k x_\ell K_{k\ell}^i \preceq 0, \quad i = 1, \dots, N,$$

where all data matrices are from \mathbb{S}^m .

4.1 The code PENBMI

User interface

The advantage of our formulation of the BMI problem is that, although non-linear, the data only consist of matrices and vectors, just like in the linear SDP case. The user does not have to provide the (first and second) derivatives of the matrix functions, as these are readily available. Hence the user interface of PENBMI is a direct extension of the interface to PENSDP described in the previous section.

In particular, the user has the choice of calling PENBMI from Matlab or from a C/C++/Fortran code. In both cases, the user has to specify the matrices $A_k^i, k = 1, \dots, n$, and $K_{k\ell}^i, k, \ell = 1, \dots, n$, for all constraints $i = 1, \dots, N$, matrix Q from the objective function and vectors $f, c, b^i, i = 1, \dots, N$. As in

the linear SDP case, all matrices and vectors are assumed to be sparse (or even void), so the user has to provide the sparsity pattern of the constraints (which matrices are present) and sparsity structure of each matrix.

Again, the most comfortable way of preparing the data and calling PENBMI is via YALMIP. In this case, the user does not have to stick to the formulation (24) and bother with the sparsity pattern of the problem—any optimization problem with vector or matrix variables and linear or quadratic objective function and (matrix) constraints will be translated by YALMIP into formulation (24) and the corresponding user interface will be automatically created. Below is a simple example of YALMIP code for the LQ optimal feedback problem formulated as

$$\begin{aligned} & \min_{P \in \mathbb{R}^{2 \times 2}, K \in \mathbb{R}^{1 \times 2}} \text{trace}(P) \\ \text{s.t.} \quad & (A + BK)^T P + P(A + BK) \prec -I_{2 \times 2} - K^T K \\ & P \succ 0 \end{aligned}$$

with

$$A = \begin{pmatrix} -1 & 2 \\ -3 & -4 \end{pmatrix}, \quad B = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Using YALMIP, the problem is formulated and solved by the following few lines

```
>> A = [-1 2;-3 -4]; B = [1;1];
>> P = sdpvar(2,2); K = sdpvar(1,2);
>> F = [P >= 0; (A+B*K)'*P+P*(A+B*K) <= -eye(2)-K'*K];
>> optimize(F,trace(P),sdpsettings('solver','penbmi'));
```

4.2 The Static Output Feedback Problem

Many interesting problems in linear and nonlinear systems control cannot be formulated and solved as LSDP. BMI formulation of the control problems was made popular in the mid 1990s [12]; there were, however, no computational methods for solving non-convex BMIs, in contrast with convex LMIs for which powerful interior-point algorithms were available.

The most fundamental of these problems is perhaps static output feedback (SOF) stabilization: given a triplet of matrices A, B, C of suitable dimensions, find a matrix F such that the eigenvalues of matrix $A + BFC$ are all in a given region of the complex plane, say the open left half-plane [3].

No LSDP formulation is known for this problem but a straightforward application of Lyapunov's stability theory leads to a BMI formulation: matrix $A + BFC$ has all its eigenvalues in the open left half-plane if and only if there exists a matrix X such that

$$(A + BFC)^T X + (A + BFC)X \prec 0, \quad X = X^T \succ 0$$

where $\prec 0$ and $\succ 0$ stand for positive and negative definite, respectively.

We present a short description of the benchmark collection *COMPl_{ib}*: the *CON*strained *Matrix*-optimization *Problem library* [20]⁶. *COMPl_{ib}* can be used as a benchmark collection for a very wide variety of algorithms solving matrix optimization problems. Currently *COMPl_{ib}* consists of 124 examples collected from the engineering literature and real-life applications for LTI control systems of the form

$$\begin{aligned}\dot{x}(t) &= Ax(t) + B_1w(t) + Bu(t), \\ z(t) &= C_1x(t) + D_{11}w(t) + D_{12}u(t), \\ y(t) &= Cx(t) + D_{21}w(t),\end{aligned}\tag{25}$$

where $x \in \mathbb{R}^{n_x}$, $u \in \mathbb{R}^{n_u}$, $y \in \mathbb{R}^{n_y}$, $z \in \mathbb{R}^{n_z}$, $w \in \mathbb{R}^{n_w}$ denote the state, control input, measured output, regulated output, and noise input, respectively.

The heart of *COMPl_{ib}* is the MATLAB function file *COMPl_{ib}.m*. This function returns the data matrices A , B_1 , B , C_1 , C , D_{11} , D_{12} and D_{21} of (25) of each individual *COMPl_{ib}* example. Depending on specific control design goals, it is possible to derive particular matrix optimization problems using the data matrices provided by *COMPl_{ib}*. A non exhaustive list of matrix optimization problems arising in feedback control design are stated in [20]. Many more control problems leading to NSDPs, BMIs or SDPs can be found in the literature.

Here we state the BMI formulation of two basic static output feedback control design problems: SOF- \mathcal{H}_2 and SOF- \mathcal{H}_∞ . The goal is to determine the matrix $F \in \mathbb{R}^{n_u \times n_y}$ of the SOF control law $u(t) = Fy(t)$ such that the closed loop system

$$\begin{aligned}\dot{x}(t) &= A(F)x(t) + B(F)w(t), \\ z(t) &= C(F)x(t) + D(F)w(t),\end{aligned}\tag{26}$$

fulfills some specific control design requirements, where $A(F) = A + BFC$, $B(F) = B_1 + BFD_{21}$, $C(F) = C_1 + D_{12}FC$, $D(F) = D_{11} + D_{12}FD_{21}$.

We begin with the SOF- \mathcal{H}_2 problem: *Suppose that $D_{11} = 0$ and $D_{21} = 0$. Find a SOF gain F such that $A(F)$ is Hurwitz and the \mathcal{H}_2 -norm of (26) is minimal.* This problem can be rewritten to the following \mathcal{H}_2 -BMI problem formulation, see, e.g. [20]:

$$\begin{aligned}\min \quad & \text{Tr}(X) \quad \text{s.t.} \quad Q \succ 0, \\ & (A + BFC)Q + Q(A + BFC)^T + B_1B_1^T \preceq 0, \\ & \begin{bmatrix} X & (C_1 + D_{12}FC)Q \\ Q(C_1 + D_{12}FC)^T & Q \end{bmatrix} \succeq 0,\end{aligned}\tag{27}$$

where $Q \in \mathbb{R}^{n_z \times n_z}$, $X \in \mathbb{R}^{n_x \times n_x}$.

⁶ See http://www.mathematik.uni-trier.de/~leibfritz/Proj_TestSet/NSDPTestSet.htm

\mathcal{H}_∞ synthesis is an attractive model-based control design tool and it allows incorporation of model uncertainties in the control design. The optimal SOF- \mathcal{H}_∞ problem can be formally stated in the following term: *Find a SOF matrix F such that $A(F)$ is Hurwitz and the \mathcal{H}_∞ -norm of (26) is minimal.* We consider the following well known \mathcal{H}_∞ -BMI version, see, e.g. [20]:

$$\begin{aligned} \min \quad & \gamma \quad \text{s.t.} \quad X \succ 0, \quad \gamma > 0, \\ & \begin{bmatrix} A(F)^T X + X A(F) & X B(F) & C(F)^T \\ B(F)^T X & -\gamma I_{n_w} & D(F)^T \\ C(F) & D(F) & -\gamma I_{n_z} \end{bmatrix} \prec 0, \end{aligned} \quad (28)$$

where $\gamma \in \mathbb{R}$, $X \in \mathbb{R}^{n_x \times n_x}$.

We present results of our numerical experiences for the static output feedback problems of *COMPlib*. The link between *COMPlib* and PENBMI was provided by the MATLAB parser YALMIP 3 [22]. All tests were performed on a 2.5 GHz Pentium with 1 GB RDRAM under Linux. The results of PENBMI for \mathcal{H}_2 -BMI and \mathcal{H}_∞ -BMI problems can be divided into seven groups: The first group consists of examples solved without any difficulties (38 problems in the \mathcal{H}_2 case and 37 problems for the \mathcal{H}_∞ setting). The second and third group contain all cases for which we had to relax our stopping criterion. In 4 (11) examples the achieved precision was still close to our predefined stopping criterion, while in 5 (7) cases deviation is significant (referring to \mathcal{H}_2 (\mathcal{H}_∞)). Then there are examples, for which we could calculate almost feasible solutions, but which failed to satisfy the Hurwitz-criterion, namely AC5 and NN10. The fourth and fifth group consist of medium and small scale cases for which PENBMI failed, due to ill conditioned Hessian of F —the Cholesky algorithm used for its factorization did not deliver accurate solution and the Newton method failed. In the \mathcal{H}_2 -setting (fourth group) these are AC7, AC9, AC13, AC18, JE1, JE2, JE3, REA4, DIS5, WEC1, WEC2, WEC3, UWV, PAS, NN1, NN3, NN5, NN6, NN7, NN9, NN12 and NN17, in the \mathcal{H}_∞ -setting (fifth group) JE1, JE2, JE3, REA4, DIS5, UWV, PAS, TF3, NN1, NN3, NN5, NN6, NN7 and NN13. The cases in the sixth group are large scale, ill conditioned problems, where PENBMI ran out of time (AC10, AC14, CSE2, EB5). Finally, for very large test cases our code runs out of memory (HS1, BDT2, EB6, TL, CDP, NN18).

4.3 Simultaneous Stabilization BMIs

Another example leading to BMI formulation is the problem of simultaneously stabilizing a family of single-input single-output linear systems by one fixed controller of given order. This problem arises for instance when trying to preserve stability of a control system under the failure of sensors, actuators, or processors. Simultaneous stabilization of three or more systems was extensively studied in [2]. Later on, the problem was shown to belong to the wide range of robust control problems that are NP-hard [3].

In [14] a BMI formulation of the simultaneous stabilization problem was obtained in the framework of the polynomial, or algebraic approach to systems control. This formulation leads to a feasibility BMI problem which, in a more general setting can be reformulated by the following procedure: Assume we want to find a feasible point of the following system of BMIs

$$A_0^i + \sum_{k=1}^n x_k A_k^i + \sum_{k=1}^n \sum_{\ell=1}^n x_k x_\ell K_{k\ell}^i \prec 0, \quad i = 1, \dots, N \quad (29)$$

with symmetric matrices $A_k^i, K_{k\ell}^i \in \mathbb{R}^{d_i \times d_i}$, $k, \ell = 1, \dots, n$, $i = 1, \dots, N$, and $x \in \mathbb{R}^n$. Then we can check the feasibility of (29) by solving the following optimization problem

$$\min_{x \in \mathbb{R}^n, \lambda \in \mathbb{R}} \lambda \quad (30)$$

$$\text{s.t.} \quad A_0^i + \sum_{k=1}^n x_k A_k^i + \sum_{k=1}^n \sum_{\ell=1}^n x_k x_\ell K_{k\ell}^i \preceq \lambda I_n, \quad i = 1, \dots, N. \quad (31)$$

Problem (30) is a global optimization problem: we know that if its global minimum λ is non-negative then the original problem (29) is infeasible. On the other hand PENBMI can only find critical points, so when solving (30), the only conclusion we can make is the following:

- when $\lambda < 0$, the system is strictly feasible;
- when $\lambda = 0$, the system is marginally feasible;
- when $\lambda > 0$ the system may be infeasible.

During numerical experiments it turned out that the feasible region of (29) is often unbounded. We used two strategies to avoid numerical difficulties in this case: First we introduced large enough artificial bounds x_{bound} . Second, we modify the objective function by adding the square of the 2-norm of the vector x multiplied by a weighting parameter w . After these modifications problem (30) reads as follows:

$$\min_{x \in \mathbb{R}^n, \lambda \in \mathbb{R}} \lambda + w \|x\|_2^2 \quad (32)$$

$$\text{s.t.} \quad -x_{\text{bound}} \leq x^k \leq x_{\text{bound}}, \quad k = 1, \dots, n$$

$$A_0^i + \sum_{k=1}^n x_k A_k^i + \sum_{k=1}^n \sum_{\ell=1}^n x_k x_\ell K_{k\ell}^i \preceq \lambda I_{n \times n}, \quad i = 1, \dots, N.$$

This is exactly the problem formulation we used in our numerical experiments.

Results of numerical examples for a suite of simultaneous stabilization problems selected from the recent literature can be found in [13].

5 PENNON

5.1 The problem and the modified algorithm

Problem formulation

In this, so far most general version of the code, we solve optimization problems with a nonlinear objective subject to nonlinear inequality and equality constraints and semidefinite bound constraints:

$$\begin{aligned}
 & \min_{x \in \mathbb{R}^n, Y_1 \in \mathbb{S}^{p_1}, \dots, Y_k \in \mathbb{S}^{p_k}} f(x, Y) \\
 & \text{subject to} \quad g_i(x, Y) \leq 0, \quad i = 1, \dots, m_g \\
 & \quad \quad \quad h_i(x, Y) = 0, \quad i = 1, \dots, m_h \\
 & \quad \quad \quad \underline{\lambda}_i I \preceq Y_i \preceq \bar{\lambda}_i I, \quad i = 1, \dots, k.
 \end{aligned} \tag{33}$$

Here

- $x \in \mathbb{R}^n$ is the vector variable
- $Y_1 \in \mathbb{S}^{p_1}, \dots, Y_k \in \mathbb{S}^{p_k}$ are the matrix variables; we denote $Y = (Y_1, \dots, Y_k)$
- f, g_i and h_i are C^2 functions from $\mathbb{R}^n \times \mathbb{S}^{p_1} \times \dots \times \mathbb{S}^{p_k}$ to \mathbb{R}
- $\underline{\lambda}_i$ and $\bar{\lambda}_i$ are the lower and upper bounds, respectively, on the eigenvalues of $Y_i, i = 1, \dots, k$

Although the semidefinite inequality constraints are of a simple type, most nonlinear SDP problems can be formulated in the above form. For instance, the problem (1) can be transformed into (33) using slack variables and equality constraints, when

$$\mathcal{A}(x) \preccurlyeq 0$$

is replaced by

$$\begin{aligned}
 \mathcal{A}(x) &= S \quad \text{element-wise} \\
 S &\preccurlyeq 0
 \end{aligned}$$

with a new matrix variable $S \in \mathbb{S}^m$.

Direct equality handling

Problem (33) is not actually a problem of type (1) that was introduced in the first section and for which we have developed the convergence theory. The new element here are the equality constraints. Of course, we can formulate the equalities as two inequalities, and this works surprisingly well for many problems. However, to treat the equalities in a “proper” way, we adopted a concept which is successfully used in modern primal-dual interior point algorithms (see, e.g., [32]): rather than using augmented Lagrangians, we handle

the equality constraints directly on the level of the subproblem. This leads to the following approach. Consider the optimization problem

$$\begin{aligned} & \min_{x \in \mathbb{R}^n} f(x) \\ & \text{subject to} \\ & \mathcal{A}(x) \preceq 0, \\ & h(x) = 0, \end{aligned} \tag{34}$$

where f and \mathcal{A} are defined as in the previous sections and $h : \mathbb{R}^n \rightarrow \mathbb{R}^d$ represents a set of equality constraints. Then we define the augmented Lagrangian

$$\begin{aligned} \bar{F}(x, U, v, p) = \\ f(x) + \langle U, \Phi_p(\mathcal{A}(x)) \rangle_{\mathbb{S}^m} + v^\top h(x), \end{aligned} \tag{35}$$

where U, Φ, p are defined as before and $v \in \mathbb{R}^d$ is the vector of Lagrangian multipliers associated with the equality constraints. Now, on the level of the subproblem, we attempt to find an approximate solution of the following system (in x and v):

$$\begin{aligned} \nabla_x \bar{F}(x, U, v, p) &= 0, \\ h(x) &= 0, \end{aligned} \tag{36}$$

where the penalty parameter p as well as the multiplier U are fixed. In order to solve systems of type (36), we apply the damped Newton method. Descent directions are calculated utilizing the factorization routine MA27 from the Harwell subroutine library ([6]) in combination with an inertia correction strategy as described in [32]. Moreover, the step length is derived using an augmented Lagrangian merit function defined as

$$\bar{F}(x, U, v, p) + \frac{1}{2\mu} \|h(x)\|_2^2$$

along with an Armijo rule.

Strictly feasible constraints

In certain applications, the bound constraints must remain strictly feasible for all iterations because, for instance, the objective function may be undefined at infeasible points [17]. To be able to solve such problems, we treat these inequalities by a classic barrier function. For this reason we introduce an additional matrix inequality

$$\mathcal{S}(x) \preceq 0$$

in problem (2) and define the augmented Lagrangian

$$\tilde{F}(x, U, p, s) = f(x) + \langle U, \Phi_p(\mathcal{A}(x)) \rangle_{\mathbb{S}^m} + s \Phi_{\text{bar}}(\mathcal{S}(x)), \tag{37}$$

where Φ_{bar} can be defined, for example, by

$$\Phi_{\text{bar}}(\mathcal{S}(x)) = -\log \det(-\mathcal{S}(x)).$$

Note that, while the penalty parameter p maybe constant from a certain index \bar{k} (see again [31] for details), the barrier parameter s is required to tend to zero with increasing k .

5.2 The code PENNON

Slack removal

As already mentioned, to transform constraints of the type $\mathcal{A}(x) \preceq 0$ into our standard structure, we need to introduce a slack matrix variable S , and replace the original constraint by $\mathcal{A}(x) = S$ element-wise, and $S \preceq 0$. Thus in order to formulate the problem in the required form, we have to introduce a new (possibly large) matrix variable and many new equality constraints, which may have a negative effect on the performance of the algorithm. However, the reformulation using slack variables is only needed for the input of the problem, not for its solution by Algorithm 1. Hence, the user has the option to say that certain matrix variables are actually slacks and these are then automatically removed by a preprocessor. The code then solves the problem with the original constraint $\mathcal{A}(x) \preceq 0$.

User interface

Unlike in the PENS DP and PENBMI case, the user has to provide not only function values but also the first and second derivatives of the objective and constraint functions. In the MATLAB and C/C++/Fortran interface the user is required to provide six functions/subroutines for evaluation of function value, gradient and Hessian of the objective function and the constraints, respectively, at a given point.

To make things simple, the matrix variables are treated as vectors in these functions, using the operator $\text{svec} : \mathbb{S}^m \rightarrow \mathbb{R}^{(m+1)m/2}$ defined by

$$\text{svec} \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ & a_{22} & \dots & a_{2m} \\ & & \ddots & \vdots \\ \text{sym} & & & a_{mm} \end{pmatrix} = (a_{11}, a_{12}, a_{22}, \dots, a_{1m}, a_{2m}, \dots, a_{mm})^T$$

In the main program, the user defines problem sizes, values of bounds, and information about matrix variables (number, sizes and sparsity patterns).

In addition, we also provide an interface to AMPL [7] which is a comfortable modelling language for optimization problems. As AMPL does not support matrix variables, we treat them, within an AMPL script, as vectors, using the operator svec defined above.

Example 1. Assume that we have a matrix variable $X \in \mathbb{S}^3$

$$X = \begin{pmatrix} x_1 & x_2 & x_4 \\ x_2 & x_3 & x_5 \\ x_4 & x_5 & x_6 \end{pmatrix}$$

and a constraint

$$\text{Tr}(XA) = 3 \quad \text{with } A = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

The matrix variable is treated as a vector

$$\text{svec}(X) = (x_1, x_2, \dots, x_6)^T$$

and the above constraint is thus equivalent to

$$x_3 + 2x_4 = 3.$$

The code needs to identify the matrix variables, their number and size. These data are included in an ASCII file that is directly read by the PENNON and that, in addition, includes information about lower and upper bounds on these variables.

5.3 Examples

Most examples of nonlinear semidefinite programs that can be found in the literature are of the form: for a given (symmetric, indefinite) matrix H find a nearest positive semidefinite matrix satisfying possibly some additional constraints. Many of these problems can be written as follows

$$\begin{aligned} & \min_{X \in \mathbb{S}^n} \frac{1}{2} \|X - H\|_F^2 \\ & \text{subject to} \\ & \quad \langle A_i, X \rangle = b_i, \quad i = 1, \dots, m \\ & \quad X \succeq 0 \end{aligned} \tag{38}$$

with $A_i \in \mathbb{S}^n$, $i = 1, \dots, m$. Probably the most prominent example is the problem of finding the nearest correlation matrix [15].

Several algorithms have been derived for the solution of this problems; see, e.g., [15, 23]. It is not our primal goal to compete with these specialized algorithms (although PENNON can solve problems of type (38) rather efficiently). Rather we want to utilize the full potential of our code and solve “truly nonlinear” semidefinite problems. In the rest of this section we will give examples of such problems.

5.4 Correlation matrix with the constrained condition number

We consider the problem of finding the nearest correlation matrix:

$$\begin{aligned} \min_X \quad & \sum_{i,j=1}^n (X_{ij} - H_{ij})^2 \\ \text{subject to} \quad & X_{ii} = 1, \quad i = 1, \dots, n \\ & X \succeq 0 \end{aligned} \tag{39}$$

We will consider an example based on a practical application from finances; see [33]. Assume that a 5×5 correlation matrix is extended by one row and column. The new data is based on a different frequency than the original part of the matrix, which means that the new matrix is no longer positive definite:

$$H_{\text{ext}} = \begin{pmatrix} 1 & -0.44 & -0.20 & 0.81 & -0.46 & -0.05 \\ -0.44 & 1 & 0.87 & -0.38 & 0.81 & -0.58 \\ -0.20 & 0.87 & 1 & -0.17 & 0.65 & -0.56 \\ 0.81 & -0.38 & -0.17 & 1 & -0.37 & -0.15 \\ -0.46 & 0.81 & 0.65 & -0.37 & 1 & -0.08 \\ -0.05 & -0.58 & -0.56 & -0.15 & 0.08 & 1 \end{pmatrix}.$$

Let us find the nearest correlation matrix to H_{ext} by solving (39) (either by PENNON or by any of the specialized algorithms mentioned at the beginning of this section). We obtain the following result (for the presentation of results, we will use MATLAB output in short precision):

```
X =
    1.0000    -0.4420    -0.2000     0.8096   -0.4585   -0.0513
   -0.4420     1.0000     0.8704   -0.3714     0.7798   -0.5549
   -0.2000     0.8704     1.0000   -0.1699     0.6497   -0.5597
    0.8096   -0.3714   -0.1699     1.0000   -0.3766   -0.1445
   -0.4585     0.7798     0.6497   -0.3766     1.0000     0.0608
   -0.0513   -0.5549   -0.5597   -0.1445     0.0608     1.0000
```

with eigenvalues

```
eigen =
    0.0000     0.1163     0.2120     0.7827     1.7132     3.1757
```

As we can see, one eigenvalue of the nearest correlation matrix is zero. This is highly undesirable from the application point of view. To avoid this, we can add lower (and upper) bounds on the matrix variable, i.e., constraints $\underline{\lambda}I \preceq X \preceq \bar{\lambda}I$. However, the application requires a different approach when we need to *bound the condition number of the nearest correlation matrix*, i.e., to add the constraint

$$\text{cond}(X) \leq \kappa.$$

This constraint can be introduced in several ways. For instance, we can introduce the constraint

$$I \preceq \tilde{X} \preceq \kappa I$$

using the transformation $\tilde{X} = \zeta X$. The problem of finding the nearest correlation matrix with a *given* condition number then reads as follows:

$$\min_{\zeta, \tilde{X}} \sum_{i,j=1}^n \left(\frac{1}{\zeta} \tilde{X}_{ij} - H_{ij} \right)^2 \quad (40)$$

subject to

$$\begin{aligned} \tilde{X}_{ii} - \zeta &= 0, \quad i = 1, \dots, n \\ I &\preceq \tilde{X} \preceq \kappa I. \end{aligned}$$

The new problem now has the NLP-SDP structure of (33). When solving it by PENNON with $\kappa = 10$, we get the solution after 11 outer and 37 inner iterations. The optimal value of ζ is 3.4886 and, after the back substitution $X = \frac{1}{\zeta} \tilde{X}$, we get the nearest correlation matrix

$$\begin{aligned} X = & \\ & \begin{array}{cccccc} 1.0000 & -0.3775 & -0.2230 & 0.7098 & -0.4272 & -0.0704 \\ -0.3775 & 1.0000 & 0.6930 & -0.3155 & 0.5998 & -0.4218 \\ -0.2230 & 0.6930 & 1.0000 & -0.1546 & 0.5523 & -0.4914 \\ 0.7098 & -0.3155 & -0.1546 & 1.0000 & -0.3857 & -0.1294 \\ -0.4272 & 0.5998 & 0.5523 & -0.3857 & 1.0000 & -0.0576 \\ -0.0704 & -0.4218 & -0.4914 & -0.1294 & -0.0576 & 1.0000 \end{array} \end{aligned}$$

with eigenvalues

$$\begin{aligned} \text{eigenvals} = & \\ & \begin{array}{cccccc} 0.2866 & 0.2866 & 0.2867 & 0.6717 & 1.6019 & 2.8664 \end{array} \end{aligned}$$

and the condition number equal to 10, indeed.

Large-scale problems

To test the capability of the code to solve large-scale problems, we have generated randomly perturbed correlation matrices H of arbitrary dimension by the commands

```
n = 500; x=10.^[-4:4/(n-1):0];
G = gallery('randcorr',n*x/sum(x));
E = 2*rand(n,n)-ones(n,n); E=triu(E)+triu(E,1)'; E=(E+E')/2;
H = (1-0.1).*G + 0.1*E;
```

For large-scale problems, we successfully use the iterative (preconditioned conjugate gradient) solver for the Newton system in Step 1 of the algorithm. In every Newton step, the iterative solver needs just a few iterations, making it a very efficient alternative to a direct solver.

For instance, to solve a problem with a 500×500 matrix H we needed 11 outer and 148 inner iterations, 962 CG steps, and 21 minutes on a notebook. Note that the problem had 125251 variables and 500 linear constraints: that means that at each Newton step we solved (approximately) a system with a *full* 125251×125251 matrix. The iterative solver (needing just matrix-vector product) was clearly the only alternative here.

We have also successfully solved a real-world problem with a matrix of dimension 2000 and with many additional linear constraints in about 10 hours on a standard Linux workstation with 4 Intel Core 2 Quad processors with 2.83 GHz and 8 Gbyte of memory (using only one processor).

5.5 Approximation by nonnegative splines

Consider the problem of approximating a one-dimensional function given only by a large amount of noisy measurement by a cubic spline. Additionally, we require that the function is nonnegative. This kind of problem arises in many application, for instance, in shape optimization considering unilateral contact or in arrival rate approximation [1].

Assume that function $f : \mathbb{R} \rightarrow \mathbb{R}$ is defined on interval $[0, 1]$. We are given its function values b_i , $i = 1, \dots, n$ at points $t_i \in (0, 1)$. We may further assume that the function values are subject to a random noise. We want to find a smooth approximation of f by a cubic spline, i.e., by a function of the form

$$P(t) = P^{(i)}(t) = \sum_{k=0}^3 P_k^{(i)}(t - a_{i-1})^k \quad (41)$$

for a point $t \in [a_{i-1}, a_i]$, where $0 = a_0 < a_1 < \dots < a_m = 1$ are the knots and $P_k^{(i)}$ ($i = 1, \dots, m$, $k = 0, 1, 2, 3$) the coefficients of the spline. The spline property that P should be continuous and have continuous first and second derivatives is expressed by the following equalities for $i = 1, \dots, m - 1$:

$$P_0^{(i+1)} - P_0^{(i)} - P_1^{(i)}(a_i - a_{i-1}) - P_2^{(i)}(a_i - a_{i-1})^2 - P_3^{(i)}(a_i - a_{i-1})^3 = 0 \quad (42)$$

$$P_1^{(i+1)} - P_1^{(i)} - 2P_2^{(i)}(a_i - a_{i-1}) - 3P_3^{(i)}(a_i - a_{i-1})^2 = 0 \quad (43)$$

$$2P_2^{(i+1)} - 2P_2^{(i)} - 6P_3^{(i)}(a_i - a_{i-1}) = 0. \quad (44)$$

The function f will be approximated by P in the least square sense, so we want to minimize

$$\sum_{j=1}^n (P(t_j) - b_j)^2$$

subject to (42),(43),(44).

Now, the original function f is assumed to be nonnegative and we also want the approximation P to have this property. A simple way to guarantee nonnegativity of a spline is to express it using B -splines and consider only nonnegative B -spline coefficients. However, it was shown by de Boor and Daniel [5] that this may lead to a poor approximation of f . In particular, they showed that while approximation of a nonnegative function by nonnegative splines of order k gives errors of order h^k , approximation by a subclass of nonnegative splines of order k consisting of all those whose B -spline coefficients are nonnegative may yield only errors of order h^2 . In order to get the best possible approximation, we use a result by Nesterov [26] saying that $P^{(i)}(t)$ from (41) is nonnegative if and only if there exist two symmetric matrices

$$X^{(i)} = \begin{pmatrix} x_i & y_i \\ y_i & z_i \end{pmatrix}, \quad S^{(i)} = \begin{pmatrix} s_i & v_i \\ v_i & w_i \end{pmatrix}$$

such that

$$P_0^{(i)} = (a_i - a_{i-1})s_i \tag{45}$$

$$P_1^{(i)} = x_i - s_i + 2(a_i - a_{i-1})v_i \tag{46}$$

$$P_2^{(i)} = 2y_i - 2v_i + (a_i - a_{i-1})w_i \tag{47}$$

$$P_3^{(i)} = z_i - w_i \tag{48}$$

$$X^{(i)} \succeq 0, \quad S^{(i)} \succeq 0. \tag{49}$$

Summarizing, we want to solve an NLP-SDP problem

$$\min_{\substack{P_k^{(i)} \in \mathbb{R} \\ i=1, \dots, m, k=0,1,2,3}} \sum_{j=1}^n (P(t_j) - b_j)^2 \tag{50}$$

subject to

$$(42), (43), (44), \quad i = 1, \dots, m$$

$$(45) - (49), \quad i = 1, \dots, m.$$

More complicated (“more nonlinear”) objective functions can be obtained when considering, for instance, the problem of approximating the arrival rate function of a non-homogeneous Poisson process based on observed arrival data [1].

Example 2. A problem of approximating a cosine function given at 500 points by noisy data of the form $\cos(4\pi \cdot \text{rand}(500, 1)) + 1 + .5 \cdot \text{rand}(500, 1) - .25$ approximated by a nonnegative cubic spline with 7 knots lead to an NSDP problem in 80 variables, 16 matrix variables, 16 matrix constraints, and 49 linear inequality constraints. The problem was solved by PENNON in about 1 second using 17 global and 93 Newton iterations.

Acknowledgements

The authors would like to thank Didier Henrion and Johan Löfberg for their constant help during the code development. The work has been partly supported by grant A100750802 of the Czech Academy of Sciences (MK) and by DFG cluster of excellence 315 (MS). The manuscript was finished while the first author was visiting the Institute for Pure and Applied Mathematics, UCLA. The support and friendly atmosphere of the Institute are acknowledged with gratitude.

References

1. F. Alizadeh, J. Eckstein, N. Noyan, and G. Rudolf. Arrival rate approximation by nonnegative cubic splines. *Operations Research*, 56:140–156, 2008.
2. V. D. Blondel. *Simultaneous stabilization of linear systems*. MacMillan, New York, 1994.
3. V. D. Blondel and J. N. Tsitsiklis. A survey of computational complexity results in systems and control. *Automatica*, 36(9):1249–1274, 2000.
4. F. J. Bonnans and A. Shapiro. *Perturbation Analysis of Optimization Problems*. Springer-Verlag New-York, 2000.
5. C. de Boor and J.W. Daniel. Splines with nonnegative b -spline coefficients. *Math. Comp.*, 28(4-5):565–568, 1974.
6. I. S. Duff and J. K. Reid. MA27—A set of Fortran subroutines for solving sparse symmetric sets of linear equations. Tech. Report R.10533, AERE, Harwell, Oxfordshire, UK, 1982.
7. R. Fourer, D. M. Gay, and B. W. Kerningham. *AMPL: A Modeling Language for Mathematical Programming*. The Scientific Press, 1993.
8. K. Fujisawa, M. Kojima, and K. Nakata. Exploiting sparsity in primal-dual interior-point method for semidefinite programming. *Mathematical Programming*, 79:235–253, 1997.
9. K. Fujisawa, M. Kojima, K. Nakata, and M. Yamashita. SDPA User’s Manual—Version 6.00. Technical report, Department of Mathematical and Computing Science, Tokyo University of Technology, 2002.
10. M. Fukuda, M. Kojima, and M. Shida. Lagrangian dual interior-point methods for semidefinite programs. *SIAM J. Optimization*, 12:1007–1031, 2002.
11. C. Geiger and C. Kanzow. *Numerische Verfahren zur Lösung unrestringierter Optimierungsaufgaben*. Springer-Verlag, 1999. In German.
12. K. C. Goh, L. Turan, M. G. Safonov, G. P. Papavassilopoulos, and J. H. Ly. Baffine matrix inequality properties and computational methods. In *Proceedings of the American Control Conference, Baltimore, MD*, 1994.
13. D. Henrion, M. Kočvara, and M. Stingl. Solving simultaneous stabilization bmi problems with pennon. LAAS-CNRS research report no. 04508, LAAS, Toulouse, 2003.
14. D. Henrion, S. Tarbouriech, and M. Šebek. Rank-one LMI approach to simultaneous stabilization of linear systems. *Systems and control letters*, 38(2):79–89, 1999.
15. N. J. Higham. Computing the nearest correlation matrix—A problem from finance. *IMA J. Numer. Anal.*, 22(3):329–343, 2002.

16. M. Kočvara and M. Stingl. PENNON—a code for convex nonlinear and semidefinite programming. *Optimization Methods and Software*, 18(3):317–333, 2003.
17. M. Kočvara and M. Stingl. Free material optimization: Towards the stress constraints. *Structural and Multidisciplinary Optimization*, 33(4-5):323–335, 2007.
18. M. Kočvara and M. Stingl. On the solution of large-scale SDP problems by the modified barrier method using iterative solvers. *Mathematical Programming (Series B)*, 109(2-3):413–444, 2007.
19. M. Kočvara and M. Stingl. On the solution of large-scale SDP problems by the modified barrier method using iterative solvers: Erratum. *Mathematical Programming (Series B)*, 120(1):285–287, 2009.
20. F. Leibfritz. *COMPlib: CONstrained Matrix-optimization Problem library* – a collection of test examples for nonlinear semidefinite programs, control system design and related problems. Technical report, University of Trier, Department of Mathematics, D-54286 Trier, Germany., 2003.
21. Y. Li and L. Zhang. A new nonlinear lagrangian method for nonconvex semidefinite programming. *Journal of Applied Analysis*, 15(2):149–172, 2009.
22. J. Löfberg. YALMIP : A toolbox for modeling and optimization in MATLAB. In *Proceedings of the CACSD Conference*, Taipei, Taiwan, 2004.
23. J. Malick. A dual approach to semidefinite least-squares problems. *SIAM J. Matrix Analysis and Applications*, 26(1):272–284, 2005.
24. H. D. Mittelmann. An independent benchmarking of SDP and SOCP solvers. *Math. Prog.*, 95:407–430, 2003.
25. J. L. Morales and J. Nocedal. Automatic preconditioning by limited memory quasi-Newton updating. *SIAM Journal on Optimization*, 10:1079–1096, 2000.
26. Y. Nesterov. Squared functional systems and optimization problems. In H. Frenk, K. Roos, and T. Terlaky, editors, *High performance optimization (Chapter 17)*, pages 405–440. Kluwer Academic Publishers, Dordrecht, 2000.
27. E. Ng and B. W. Peyton. Block sparse cholesky algorithms on advanced uniprocessor computers. *SIAM Journal on Scientific Computing*, 14:1034–1056, 1993.
28. J. Nocedal and S. Wright. *Numerical Optimization*. Springer Series in Operations Research. Springer, New York, 1999.
29. D. Noll. Local convergence of an augmented lagrangian method for matrix inequality constrained programming. *Optimization Methods and Software*, 22(5):777–802, 2007.
30. R. Polyak. Modified barrier functions: Theory and methods. *Mathematical Programming*, 54:177–222, 1992.
31. M. Stingl. *On the Solution of Nonlinear Semidefinite Programs by Augmented Lagrangian Methods*. PhD thesis, Institute of Applied Mathematics II, Friedrich-Alexander University of Erlangen-Nuremberg, 2006.
32. A. Wächter and L. T. Biegler. On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. *Math. Prog.*, 106:25–57, 2006.
33. R. Werner and K. Schöttle. Calibration of correlation matrices—SDP or not SDP. Submitted, 2010.
34. S. Wright. *Primal-Dual Interior-Point methods*. SIAM, 1997.